Towards Methodological Guidance for User Interface Development Life Cycle

Francisco Javier Cano Muñoz[1], Jean Vanderdonckt[2]
[1]Prodevelop S.L., 46001 Valencia, Spain – fjcano@prodevelop.es
[2]Université catholique de Louvain, Louvain School of Management
Louvain Interaction Laboratory, Place des Doyens, 1 – B-1348 Louvain-la-Neuve (Belgium)
jean.vanderdonckt@uclouvain.be – Phone: +32 10 478525

ABSTRACT
This paper describes how methodological guidance could be provided to user interfaces designers throughout user interface development life cycle supported when a model-driven engineering is involved. For this purpose, a methodologist firstly creates a dashboard model according to a corresponding meta-model in order to define a user interface development path that consists of a series of development tasks (that structure the development path into development actions) and dependencies (that serve as methodological milestones). Once created, a user interface designer enacts a previously defined user interface development path by instantiating and interpreting a dashboard model while being provided with methodological guidance to conduct this development path. This guidance consists of steps, sub-steps, cheat sheets, and methodological actions.

Author Keywords
Dashboard model, dependency, development path, method enactment, method engineering, methodological guidance, user interface development life cycle.

General Terms
Design, Experimentation, Human Factors, Verification.

ACM Classification Keywords
D2.2 [**Software Engineering**]: Design Tools and Techniques – *Modules and interfaces; user interfaces*. D2.m [**Software Engineering**]: Miscellaneous – Rapid Prototyping; reusable software. H5.2 [**Information interfaces and presentation**]: User Interfaces – *Prototyping.*.

INTRODUCTION
When User Interface (UI) designers, modelers, analysts, graphical designers, and developers are given the opportunity to rely on a Model-Driven Engineering (MDE) software environment to produce a UI, they often complain on the lack of methodological guidance throughout the *UI development life cycle* (UIDLC) provided by MDE:

- Although MDE explicitly relies on a structured transformation process, namely involving model-to-model transformation (M2M) and model-to-code compilation (M2C), designers do not easily perceive where some degree of freedom suggests alternative choices in the UIDLC and where some degree of determinism constraints these choices. MDE is often considered as a straightforward approach, if not sequential, where little or no degree of freedom is offered, even when multiple *development paths* are possible [15].
- Facing the multiplicity of models (e.g., task, domain, abstract UI, concrete UI, context) in a particular development path (e.g., forward engineering, reverse engineering, round-trip engineering), the designer is rarely provided with some guidance on when and how to produce such models [13].
- When a particular step in the UIDLC should be conducted, designers do not determine easily which software should be used for this purpose, especially when different software support the same step, partially or totally. When a particular software is selected, they often feel lost in identifying the right actions to execute in order to achieve the step in the UIDLC [1].
- The multiplicity of development paths conducted among or within various organizations, in particular software development companies [3], increases the feeling of applying a UIDLC that remains not explicitly supported and that requires extensive training to become effective and efficient.
- Although several standardization efforts (e.g., the international standard for describing the method of selecting, implementing and monitoring the software development life cycle is ISO 1220707) and official organizations promote the usage of *process models* in order to increase the productivity of the development life cycle and the quality of the resulting software, they do not often rely on an explicit definition and usage of a method in these process models.

The above observations suggest that MDE is often more driven by the software intended to support it, less by the models involved in the UIDLC, and even less by a method that is explicitly defined to help UI designers. Lao Tch'ai Tche, an old Chinese philosopher, expressed the need for method in the following terms: some consider it noble to have a method; other consider it noble not to have a method; not to have a method is bad; but to stop entirely at any method is worse still; one should at first observe rules severely, then change them in an intelligent way; the aim of possessing method is to seem finally as if one had no method.

Therefore, we believe that a UIDLC according to MDE should rest on three pillars in a balanced way: *models* that capture the various UI abstractions required to produce a UI, a *method* that defines a methodological approach in order to proceed and ensure an appropriate UIDLC, and a *software* support that explicitly supports applying the method. For this purpose, the remainder of this paper is structured as follows: Section 2 presents a characterization of these three pillars in order to report on some initial pioneering work conducted in the area of UI method engineering with the particular emphasis of methodological support. Section 3 introduces the dashboard model as a mean to define a method that may consist of one or many development paths by defining its semantics and syntax. Section 4 describes how a method could be enacted, i.e. how a development path can now be applied for a particular UI project by interpreting the dashboard model. Section 5 provides a qualitative analysis of the potential benefits of using this dashboard model for method engineering in the UIDLC. Section 6 discusses some avenues of this work and presents some conclusion.

RELATED WORK

In general in computer science, a Software Development Life Cycle (SDLC) is the structure imposed on the software development by a development method. Synonyms include software development and software process. Similarly, in the field of UI, a UI development life cycle (UIDLC) consists of the development path(s) defined by a UI development method in order to develop a UI (Fig. 1). Representative examples of include: the Rational Unified Process (RUP) or the Microsoft Solution Framework (MSF). Each development path is recursively decomposed into a variety of development steps that take place during the development path. Each step uses one or several models (e.g., task, domain, and context) and may be supported by some software. All pieces of software, taken together support the development method.

For instance, the development path "Forward engineering" may be decomposed in to a series of development steps: building a task model, building a domain model, building a context model, linking them, producing a UI model from these model, then generate code according to M2C. *Method engineering* [1] is the field of defining such development methods so that a method is submitted to *method configuration* [9] when executed.

The meta-method Method For Method Configuration (MMC) [9] and the Computer-Aided Method Engineering (CAME) tool MC Sandbox [10] have been developed to support method configuration. One integral part of the MMC is the method component construct as a way to achieve effective and efficient decomposition of a method into paths and paths into steps and sub-steps and explain the rationale that exist behind this decomposition. Method engineering has already been applied to various domains of computer science such as, but not limited to: information systems [1], collaborative applications [12], and complex systems [6]. Typically, method engineering is based on a meta-model [7,8] and could give rise to various adaptations, such as situational method engineering [6] and method engineering coupled to activity theory [10]. In Human-Computer Interaction (HCI), we are not aware of any significant research and development on applying method engineering to the problem of engineering interactive systems. Several HCI development methods do exist and are well defined, such as a task-based development method [16], method-user-centered design [9], activity theory [10], but they are not expressed according to method engineering techniques, so they do not benefit from its potential advantages.

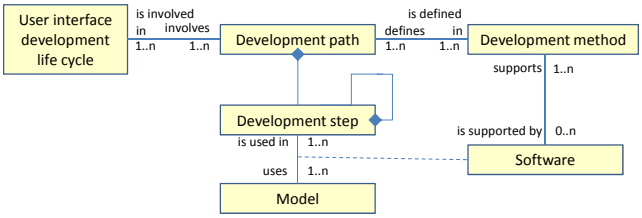

Figure 1. Structure of a UI development life cycle.

Probably the first one to address method engineering in HCI was the MIDAS (Managing Interface Design via Agendas/Scenarios) [11] environment. In this software, a methodologist was able to define a method by its different paths that could be followed and the steps required for achieving each path. MIDAS was able to show at any time when a method is executed, what are the different paths possible (e.g., design alternatives, criteria) by looking at design intentions stored in a library. MIDAS is tailored to the HUMANOÏD environment [11] and does not rely on a meta-model for defining a method and to execute. But it was a real methodological help.

User Interface Description Languages (UIDLs) do not possess any methodological guidance based on method engineering because they mostly concentrate on the definition and the usage of their corresponding syntaxes and less on the definition of the method [4].

MUICSER [5] provides mechanisms for expressing scenarios with personas based on storyboards. As such, they also used models through a storyboard software. Therefore, it could also benefit from the potential advantages of method engineering. A more recent effort used Service Oriented Architecture (SOA) to define and enact a method, but there was no real software for achieving the method engineering.

In conclusion, very few works exist on applying method engineering to HCI, but several existing work could benefit from it.

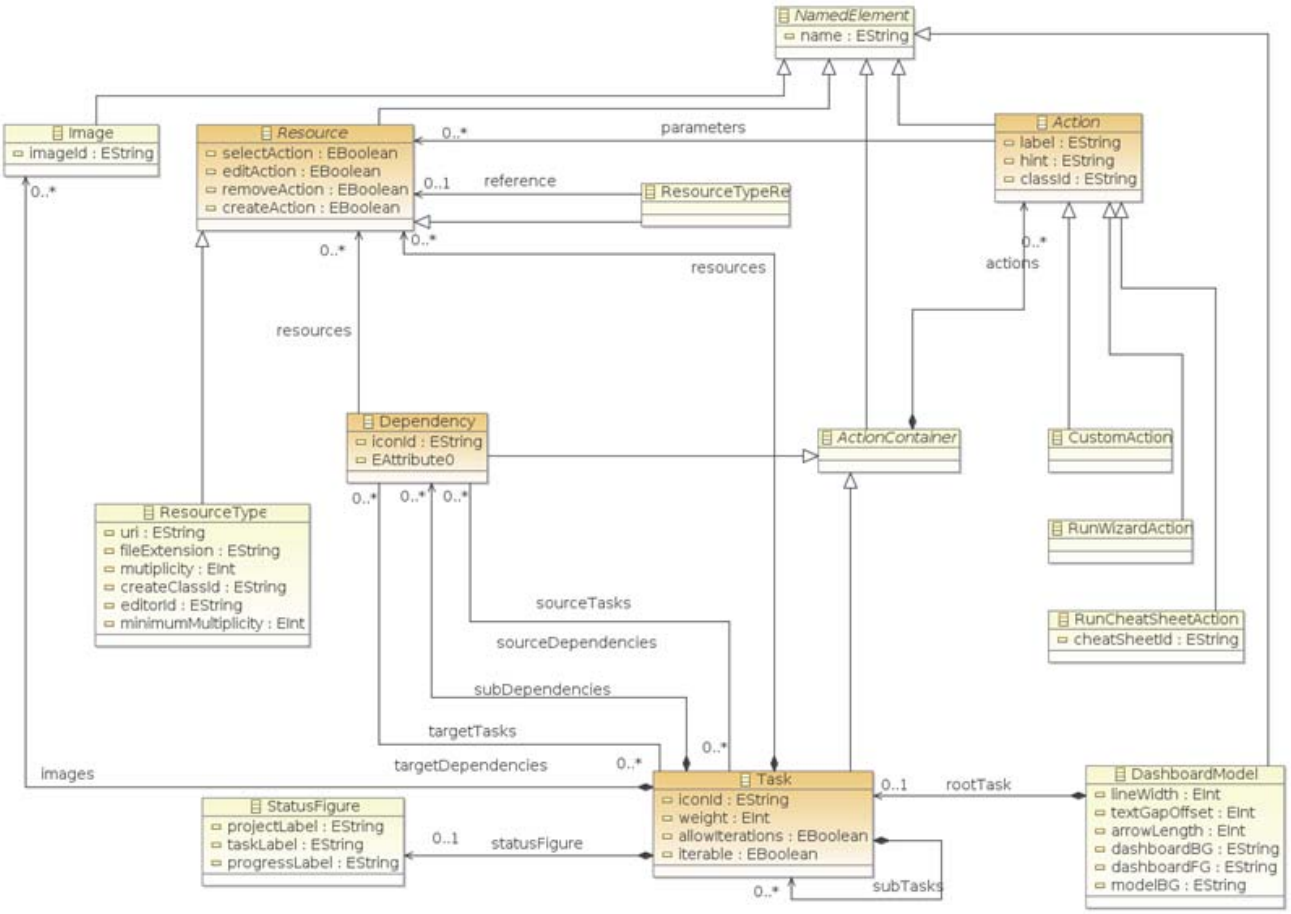

Figure 2. The Moskitt meta-model for a methodological dashboard.

A META-MODEL FOR A METHODOLOGICAL DASHBOARD

To adhere to method engineering principles, a meta-model is defined [8] that addresses its methodological concepts as outlined in Fig. 2. The dashboard is based on a metamodel that allows the description of development steps via their decomposition in Tasks, Resources required in Tasks and Dependencies between Tasks. This Dashboard metamodel has been expressed using Ecore/Eclipse Modelling Framework (EMF) and implemented in the MOSKitt environment [14]. The complete metamodel can be found in this repository URL open to the public at http://subversion. moskitt.org/gvcase-gvmetrica/dashboard/trunk/es.cv. gvcase.mdt. dashboard/model/. The main entities, i.e. Task, Resource, Dependency and Action, are structured as follows:

NamedElement: consists of a common ancestor for all metamodel elements. With the experience of the definition of several metamodels (more than 10) in the MOSKitt environment we have found very useful to have a common ancestor element that all other elements in the metamodel inherit from. It simplifies several tasks in the following steps in the MDE approach we follow, such as allowing to identify whether any given element belongs to this metamodel by checking its ancestry, and providing several properties we need in all elements of our metamodel, such as the 'name' property.

- name: EString → this element's name.

DashboardModel: represents a complete development path and at the same time is the root element of the metamodel. It holds the visual configuration to be used in the interpreter/enactment view.

- lineWidth: EInt → border elements' width to be used when the dashboard model is shown in the interpreter/enactment view.
- textGapOffset: EInt → gap between text when the dashboard model is shown in the interpreter/enactment view.
- arrowLength: EInt → length of arrows when the dashboard model is shown in the interpreter/enactment view.
- dashboardBG: EString → background color to be used when the dashboard model is shown in the interpreter/enactment view.

- dashboardFG: EString → foreground color to be used when the dashboard model is shown in the interpreter/enactment view.
- modelBG: EString → backgound color to be used in the interpreter/enactment view for tasks and dependencies.
- RootTask: Task [0..1] → reference to the whole process that is represented as a task.

Task: represents one development step of the development path. A Task will always be bounded by Dependencies, except for the Tasks involving the first and last steps of the process. A Task can produce or consume zero or many Resources. As an ActionContainer, a Task can perform Actions on selected Resources.

- iconId: EString → identifier of the icon that represents this Task when seen in the interpreter/enactment view.
- weight: EInt → this Task's weight in the development path (Process) in absolute value.
- allowIterations: EBoolean → indicates whether this Task allows more than one iteration of it to be created.
- iterable: EBoolean → indicates whether this Task can be iterated.
- subTasks: Task [0..*] → Tasks representing the different steps to perform this Task.
- resources: Resource [0..*] → Resources produced or consumed by this Task.
- sourceDependencies: Dependency [0..*] → Dependencies that must be clean before this Task can be started.
- targetDependencies: Dependency [0..1] → Dependencies that can be cleaned when this Task is complete.
- subDependencies [0..1] → Dependencies that are contained in this Task due to it having more than one internal development sub-steps.
- statusFigure: StatusFigure [0..1] → figure to be shown in the interpreter/enactment view regarding the advancement of this Task.
- Images: Image [0..*] → images contained in this Task to be shown in the interpreter/enactment view.

Dependency: represents a milestone in the development path, which means that a series of development steps should be achieved before proceeding to the next development step. The Milestone is here introduced as a straightforward mechanism to synchronize different types of development steps, whatever their purpose is. Each Dependency is a step in the development path (Process) that forces the preceding Tasks to synchronize. A Dependency can require zero or more Resources from previous Tasks to be completed. As an ActionContainer, a Dependency can perform one or more Actions on selected Resources.

- resources: Resource [0..*] → Resources this Dependency will produce or consume. Usually but not necessarily these Resources will be ResourceTypeRefs referencing Resources from previous Tasks.
- sourceTasks: Task [0..*] → preceding Tasks that use this Dependency as their Milestone.
- targetTasks: Task [0..*] → Tasks that need this Dependency to be cleaned before they can be performed.

Resource: consists of a material or immaterial entity, produced or consumed by a Task or a Dependency of this development path (Process). Resources represent from model definition files to metamodel, document to PDF files.

- selectAction: EBoolean → indicates whether the 'select' action is available for this Resource in the ResourceManagement dialog in the interpreter/enactment view.
- editAction: Eboolean → indicates whether the 'edit' action is available for this Resource in the ResourceManagement dialog in the interpreter/enactment view.
- removeAction: Eboolean → indicates whether the 'remove' action is available for this Resource in the ResourceManagement dialog in the interpreter/enactment view.
- createAction: Eboolean → indicates whether the 'create' action is available for this Resource in the ResourceManagement dialog in the interpreter/enactment view.

ResourceType: represents the actual type of the Resource element, as in the kind of model file or file document format.

- uri: EString → URI pointing to the real resource.
- fileExtension: EString → extension of the files this ResourceType represents.
- multiplicity: Eint → maximum multiplicity of real resources this Resource can hold.
- minimumMultiplicity: EInt → minimum multiplicity of real resources this Resource can hold.
- createClassId: EString → identifier of the factory that will allow the creation of this kind of resource via the ResourceManager in the interpreter/enactment view.
- editorId: EString → identifier of the editor that allows the edition of this kind of resource via the ResourceManager in the interpreter/enactment view.

ResourceTypeRef: represents a reference to a Resource. It is to all effects equal to a Resource except that the Re-

sourceManager in the interpreter/enactment view does not allow its modification in any way.

- reference: Resource [0..1] → referenced Resource.

Action: represents an action to be performed by the user when enacting the process. An Action can range from launching a transformation to opening a cheatsheet to visiting a web page.

- label: EString → human readable label to be shown in the interpreted/enactment view.
- hint: Estring → additional information to be given to the Action as a parameter, such as an identifier, a message, etc.
- classId: Estring → identifier of the factory that allows the creation and execution of this Action.

ActionContainer: represents any element in the metamodel that can hold and perform Actions.

CustomAction: represents a custom Action allows the methodologist to specify uncommon Actions with an external specification of the Action.

RunWizardAction: expresses a specialized Action that runs the wizard specified by the hint parameter of the Action.

RunCheatSheetAction: expresses specialized Action that shows the user a guide or cheatsheet.

- cheatSheetId: EString → identifier of the cheatsheet to show.

StatusFigure: allows selecting an image to be shown in the interpreter/enactment view which provides the level of progress in the process.

Image: represents a decorative image that will be shown in the interpreter/enactment view.

METHOD DEFINITION AND ENACTMENT

In order to define a UI development method based on one or many UI development paths (e.g., simplified, enhanced forward engineering, forward engineering with loops) as defined in Fig. 1, the person who is responsible for defining such a methodology has to create one Dashboard model based on the meta-model outlined in Fig. 2. A Dashboard model therefore represents the definition of a particular development path, but may also contain several development paths in one model thanks to the concept of milestone.

A milestone consists of a synchronization points between tasks (e.g., development steps) involved in a development path and is attached to a *synchronization condition*. Such a condition governs the contribution of each task to the milestone (e.g., AND, OR, XOR, NOT, *n* iterations). Once the synchronization condition is satisfied, the milestone is considered to be achieved and the development path can proceed to the next task (development step).

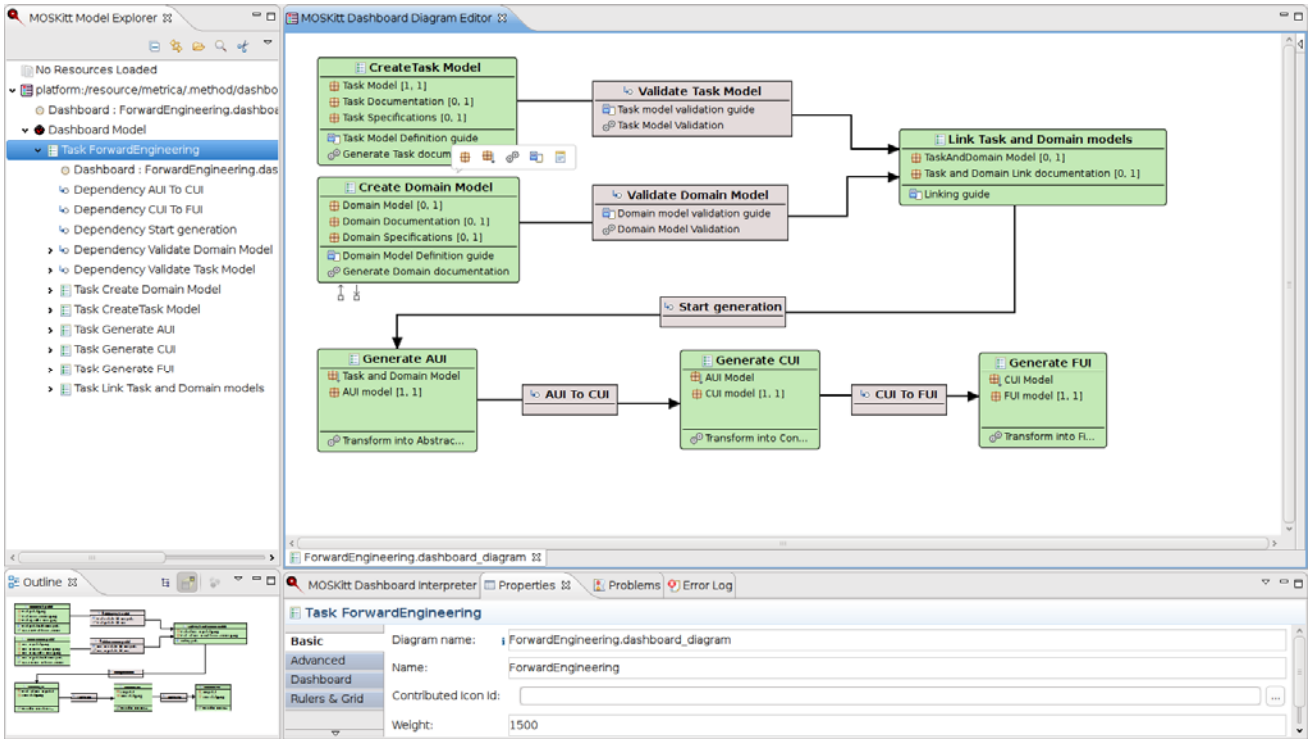

Figure 3. The Dashboard model for the "Forward engineering" development path.

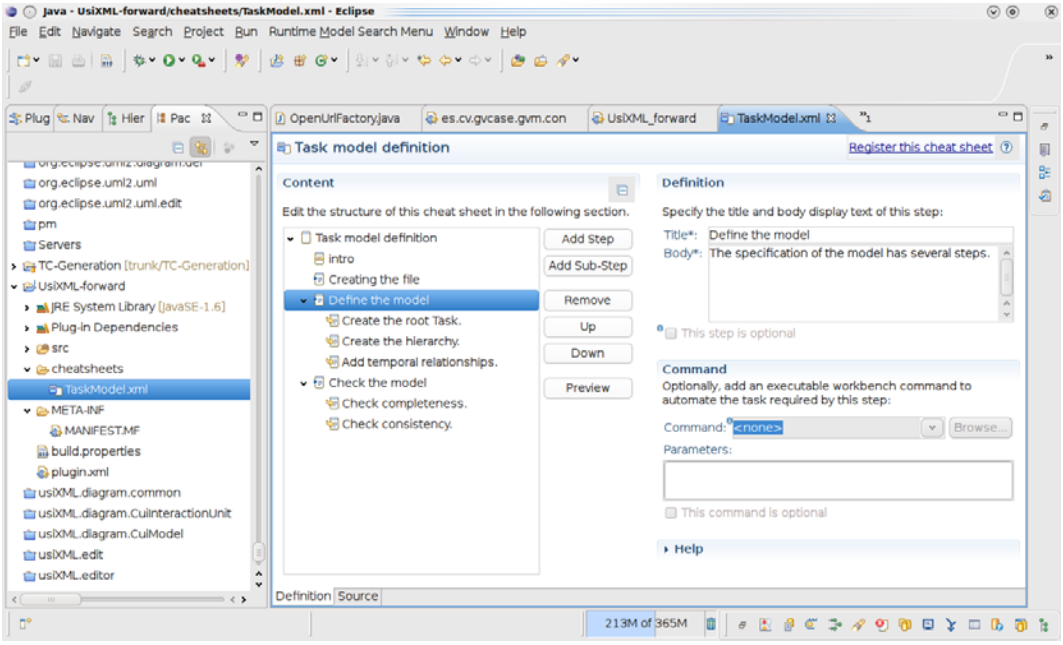

Figure 4. Cheatsheet for providing methodological guidance on task model definition.

Fig. 3 depicts in Moskitt how a Dashboard model is created for the development path "Forward Engineering" that consists of the following development steps (that are represented as tasks to achieve to complete the development step):

- *Create Task Model*: this task is aimed at creating a task model that is compliant with the task meta-model, whatever the task meta-model would be. This task manipulates three resources:
  1. One and only one task model that will result from this task.
  2. An optional document containing a documentation of the task modeled.
  3. An optional set of task formal specifications.

  A "task model definition guide" is a cheatsheet provided for giving methodological guidance on how to define a task model. Fig. 4 details some potential development steps and sub-steps for this purpose in a cheatsheet. A *cheatsheet* is hereby referred to as a methodological panel that is provided from the methodologist to the method applier with any rules, heuristics, principles, algorithms, or guidelines that are helpful for achieving the associated task (here, creating a task model that is correct, complete, and consistent). An action "Generate Task Documentation" is added in order to specify a task model would ultimately result from it. For this task, Fig. 5 shows the relevant parameters, including the file extension of the file to be created. In this case, we could specify one file extension (e.g., ".usi" for a UsiXML compliant file [15]) or ".CTT" for a task model expressed according to the ConcurTaskTree notation). In this way, when the method will be enacted, the Dashboard will automatically launch the model editor corresponding to the file extension by default or selected by the method applier (Fig. 6).
- *Validate Task Model*: once the task model has been created, its validity with respect to its corresponding task meta-model is checked by means of Eclipse model checking techniques. Therefore, only one action is triggered: "Task model validation". Note that this task serves as a milestone: the method applier cannot proceed with the next tasks if the synchronization condition (here the availability of a valid task model) is not satisfied.
- *Create Domain Model*: this task is aimed at creating a domain model that is compliant with the task meta-model, whatever the task meta-model would be. It contains three resources, one cheatsheet and one action that are similar to those introduced for the task model.
- *Validate Domain Model*: once the domain model has been created, its validity with respect to its corresponding domain meta-model is checked by means of Eclipse model checking techniques.
- *Link Task and Domain models*: this task is aimed at establishing a link from the nodes of a task model to the appropriate nodes of a domain model thanks to the set of mappings accepted between these two models (e.g., a task observes a domain class, a task supports input/output of a set of attributes taken from different classes, a task triggers a method belonging to a class). Note that there is a dependency between this task and the two previous ones in order to ensure that the linking will be applied on two syntactically valid task and domain models.
- *Milestone: start the Abstract UI generation*: when the task model has been linked to a domain model, we have all the elements in order to initiate a generation of an Abstract UI [15]. Again, this serves as a milestone.
- *Generate AUI*: this task is aimed at (semi-)automatically generating an Abstract UI (AUI). For this purpose, an input resource "Task and domain models linked" (coming from the previous milestone) will result into an output resource "AUI model" by means of the action "Transform into AUI". This action is related to a set of

transformation rules that are automatically applied to the input resource in order to obtain the output resource. Again, the embedded transformation engine in Eclipse could be used for this purpose or a custom transformation engine could be specified based on a file extension. In this definition, only one set of transformations is defined, but several alternative sets of transformation rules could be considered, thus leaving the control to the method applier by selecting at run-time which set to apply. Furthermore, this action is related to a transformation step (here, a M2M), but it could also be attached to an external algorithm that is programmed in a software. When all these alternatives coexist, a cheatsheet could be added to help the method applier in selecting an appropriate technique for ensuring this action (e.g., a transformation or an external algorithm) and parameters that are associated to this actin (e.g., a particular transformation set).

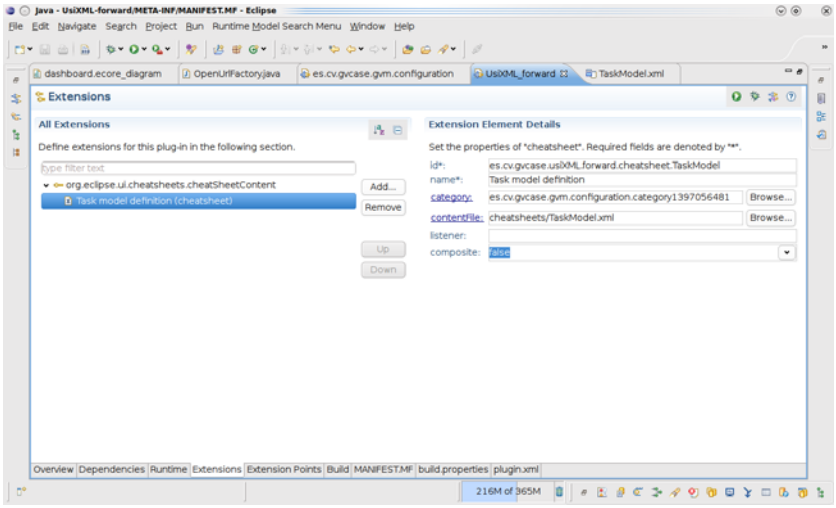

Figure 6. Definition of file extension and associated software, e.g., through a cheatsheet.

- *Milestone "AUI to CUI"*: this milestone serves as a synchronization point for initiating the next development step through the task required for this purpose.
- *Generate CUI*: this task is similar to the "Generate AUI" except that a CUI is produced instead of a AUI, but with parameters that govern the CUI generation.
- *Milestone "CUI to FUI"*: this milestone serves for initiating the last step.
- *Generate FUI*: this task is aimed at transforming the CUI resulting from the previous task into code of the Final UI (FUI) by means of M2C transformation. Again, we may want to specify here that the transformation could be achieved by code generation or by interpretation of the CUI model produced. In the first case, a code generator is executed while a FUI interpreter renders the CUI into a FUI in the second case. Again, one default interpreter could be specified or the method applier can pick another one from a list of potential interpreters or rendering engines.

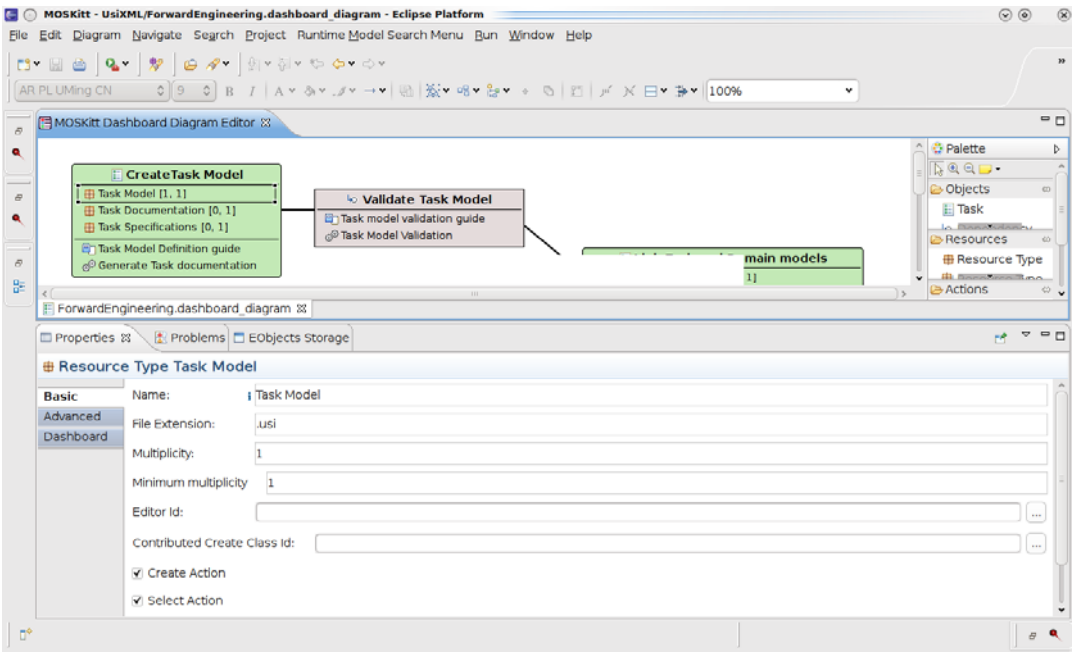

Figure 5. The file extension associated to a resource in the Dashboard model.

It is important to state that the **dashboard model is independent of any method, any metamodel and any User Interface Description Language (UIDL)**. It could be used for defining any UIDLC, any method that supports UIDLC (such as [9,12,16] to name a few), any metamodel of a model involved in such a UIDLC, and any UIDL (see [4] for a list of some representative examples). The only requirement is that each model should be explicitly linked to its corre-

sponding meta-model in order to check its validity and conformity with respect to the meta-model as it is typically the case in MDA. Transformations gathered in transformation steps should satisfy the same requirement, unless they are executed outside the Eclipse platform. The advantage of this approach is that all models and transformations between are defined by their corresponding meta-models in Eclipse, but forces to define them beforehand.

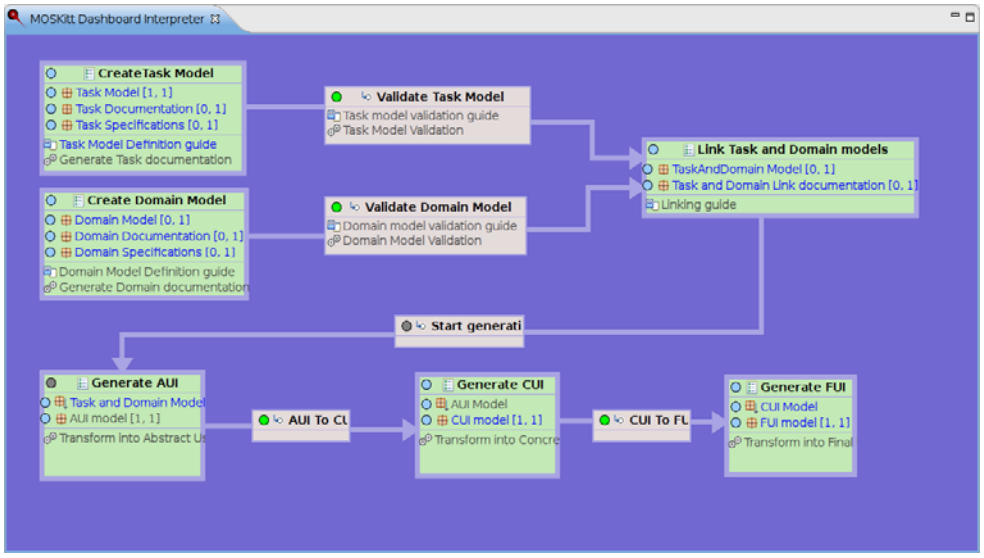

Figure 7. Definition of file extension and associated software, e.g., through a cheatsheet.

Once one or several development paths of UI development method have been defined in a dashboard model, the method can be *enacted* [2] by instantiating the dashboard model. This instantiation results into a run-time representation of the Dashboard (Fig. 7) that depicts the progression of tasks already achieved, future and pending tasks, all with their associated resources. For instance, if a task requires to output resources to be created, this task will only be considered finished when the corresponding actions will have been able to produce the required resources. The method enactment is then under the responsibility of the person who is in charge of applying the method defined, e.g. an analyst, a designer. In the next section, we review potential benefits brought by the MDA approach under the light of this dashboard approach.

QUALITATIVE EVALUATION

Usually, potential benefits that can be expected from MDE can be summarized as:

1. Benefits resulting from the existence of a design phase:

   — *Reducing the gap between requirements and implementation*: A design phaseaims to ensure in advance that the implementation really addresses the customer's and user's requirements. The dashboard largely contributes to this by explicitly defining the output of a development task that should serve as an input for a next development task, whatever the development steps are.

   — *Stakeholder coordination*: Previous planning of the UIDLC enables the stakeholders (e.g., designers, developers, testers) to coordinate their work e.g. by dividing the system into several parts and defining mappings between them.

   — *Well-structured systems*: A design phase provides explicit planning of the system architecture and the overall code structure. This facilitates implementation itself as well as maintenance.

2. Benefits resulting from the use of visual abstract models:

   — *Planning on adequate level of abstraction*: Modeling languages provide the developer concepts for planning and reasoning about the developed system on an adequate level of abstraction. The Dashboard is based on a meta-model (Fig. 2).

   — *Improved communication by visual models*: The visual character of modeling languages can lead to increased usability (understanding, perceiving, exploring, etc.,) of design documents for both author and other developers. The Dashboard model is itself entirely visual, which is particularly important for representing the progression of the method enactment.

   — *Validation*: (Semi-)Formal modeling languages enable automatic validation of the design.

   — *Documentation*: Models can be used as documentation when maintaining the system.

   — *Platform-independence*: Platform-independent models can be reused or at least serve as starting point when implementing the system for a different platform. This includes development platforms like a programming language or component model, as well as deployment platforms like the operating system or target devices. As said, the dashboard based approach is independent of any method, model and UIDL, thus supporting any UIDLC in principle.

3. Benefits resulting from code generation: the benefits associated to this appear when the method is enacted.

- *Enhanced productivity*: Generating code from a given model requires often only a teeny part of time compared to manual mapping into code.
- *Expert knowledge can be put into the code generator*: Expert knowledge – e.g. on code structuring, code optimizations, or platform-specific features – can once be put into the code generator and then be reused by all developers.
- *Reduction of errors*: Automatic mapping prevents from manual errors.

4. Meta goals:

   - *Easier creation and maintenance of development support*: the dashboard (rein)forces the method applier to stick to the initial definition of the method. Therefore, if any deviation with respect to this definition should be recorded, it should be introduced as an exception of the method enactment.
   - *Knowledge about creation of modeling languages*: MDE concepts and definitions reflect existing knowledge about modeling, modeling languages, and code generation. The dashboard does not escape from this since it is itself based on a meta-model that could be instantiated at any time.
   - *Frameworks and tools*: Tools like Eclipse Modeling Tools provide sophisticated support for all steps in MDE like creating and processing metamodels, creating modeling editors, and defining and executing transformations. The Dashboard is implemented in the Moskitt environment [14] that is itself on top of Eclipse.

5. Maintenance of modeling language and transformations:

   - *Systematic and explicit definition of metamodels and transformations facilitates*
   - *Maintenance of modeling languages and code generators*: modeling language, associated model-to-model transformations and model-to-code compilations can be maintained at a level of expressiveness that is higher than a traditional programming or markup language.
   - *Reuse of metamodels and transformations*: MDE compliant explicit metamodels and transformations can be understood and reused by others.

One the one hand, the method defined in a dashboard can provide substantive and effective knowledge, such as methodological guidance, on how to enact the method.

But on the other hand, once this method is defined, it is almost impossible to break the frontiers imposed by this method. The drawback of this is that any deviation triggers a round-trip engineering problem: the method that was enacted imposes modifying its corresponding dashboard model and propagating the changes in a new enactment of the new method, which is a hard procedure. This could be perceived as a burden by stakeholders who could feel that they are forced to enter in the dashboard everything that is required to properly conduct the method. But this information is intrinsically expressed in a consistent format that could give rise to a library of method definitions.

The dashboard approach has been applied to different real-world case studies, including a large-scale project by the Conselleria de Infraestructuras y Transporte (Valencia, Spain, Figure 8).

CONCLUSION

In this paper, we presented the dashboard model as a way to support the method engineering of a user interface development life cycle. For this purpose, we first defined what such a development life cycle is and how to structure it according to the principles of method engineering [1,2]. This development life cycle is then expressed in terms of the following concepts: one or several development steps are defined in one single dashboard in order to create one development method, a development (sub-)step becomes a task to be achieved in the dashboard, the models involved in a development step become resources to be created and consumed by a task in the dashboard, the software required to manipulate these models become associated to resources via their associated file extension and/or from a list of potential software (e.g., model editor, model validator, model checker, transformation engine).

The next step of this research will consider the forthcoming ISO 24744 standard on method engineering [2] that defines a set of concepts that support the definition and the enactment of a method based on well-defined concepts along with a graphical notation that combines structural aspects (e.g., how a task is decomposed into sub-tasks) and temporal aspects (e.g., how tasks are related to each other through dependencies and constraints).

ACKNOWLEDGMENTS

The authors would like to acknowledge of the ITEA2-Call3-2008026 USIXML (User Interface extensible Markup Language) European project and its support by Région Wallonne DGO6.

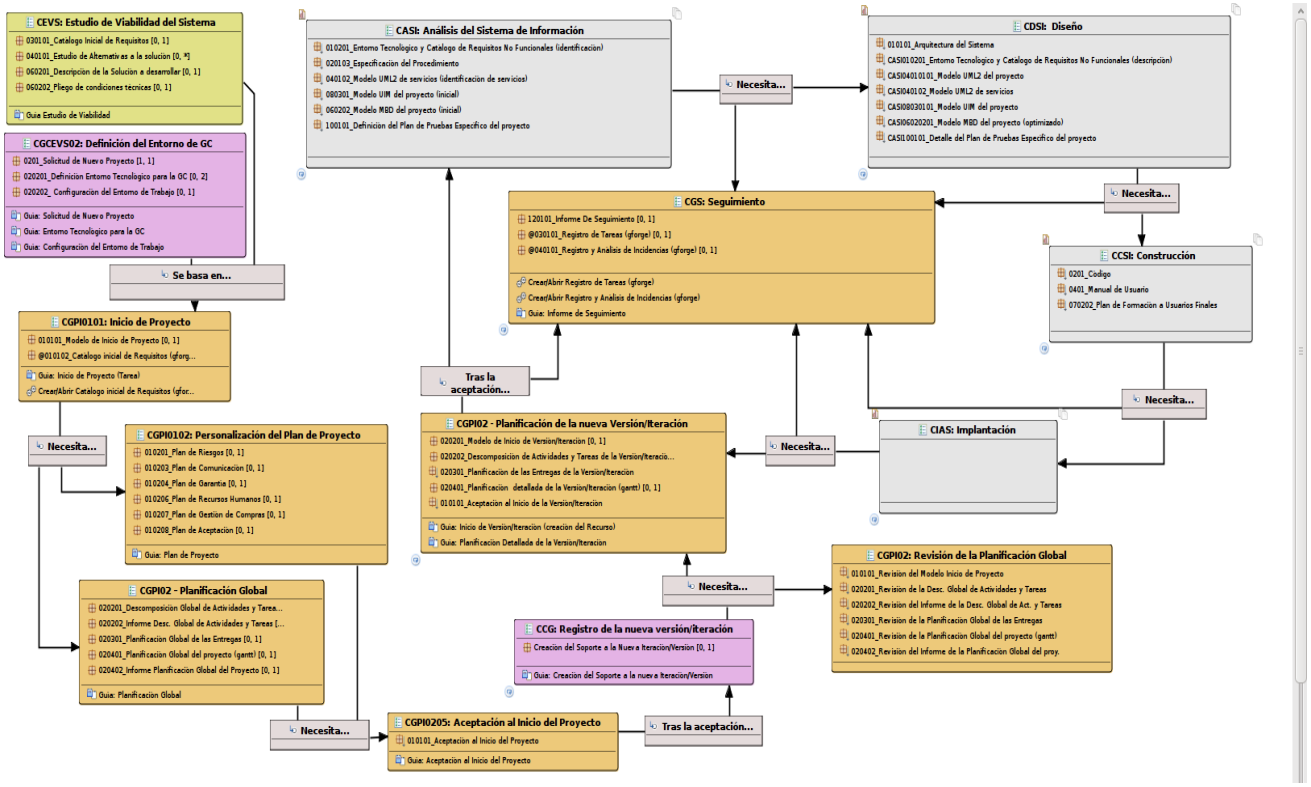

Figure 8. Dashboard model for Spanish Concil.

REFERENCES

1. Göransson, B., Gulliksen, J., Boivie, I. The usability design process - integrating user-centered systems design in the software development process. *Software Process: Improvement and Practice, 8*, 2 (2003), pp. 111-131.
2. Brinkkemper, S. Method engineering: engineering of information systems development methods and tools. *Information and Software Technology, 38*, 4 (1996), pp. 275–280.
3. Gonzalez-Perez, C. and Henderson-Sellers, B. A work product pool approach to methodology specification and enactment. *Journal of Systems and Software, 81*, 8 (2008), pp. 1288-1305.
4. Griffiths, T., Barclay, P.J., Paton, N.W., McKirdy, J., Kennedy, J.B., Gray, P.D., Cooper, R., Goble, C.A., and Pinheiro da Silva, P. Teallach: a model-based user interface development environment for object databases. *Interacting with Computers 14*, 1 (2001), pp. 31-68.
5. Guerrero García, J., González Calleros, J.M., Vanderdonckt, J., and Muñoz Arteaga, J. A Theoretical Survey of User Interface Description Languages: Preliminary Results. In: *Proc. of LA-Web/CLIHC'2009* (Merida, November 9-11, 2009). IEEE Computer Society Press, Los Alamitos (2009), pp. 36–43.
6. Henderson-Sellers, B. and Ralyté, J. Situational Method Engineering: State-of-the-Art Review. *Journal of Universal Computer Science, 16*, 3 (2010), pp. 424-478.
7. Hug, Ch., Front, A., Rieu, D., and Henderson-Sellers, B. A method to build information systems engineering process metamodels. *J. of Systems and Soft. 82*, 10 (2009), pp. 1730-1742.
8. Jeusfeld, M.A., Jarke, M., and Mylopoulos, J. Metamodeling for Method Engineering. The MIT Press, New York (2009).
9. Karlsson, F. and Ågerfalkb, P.J. Method-User-Centred Method Configuration. In: Ralyté, J., Ågerfalk, P.J., Kraiem, N. (Eds.), *Proc. of Situational Requirements Engineering Processes SREP'05* (Paris, August 29–30, 2005)
10. Karlsson, F. and Wistrand, K. Combining method engineering with activity theory: theoretical grounding of the method component concept. *European Journal of Information Systems, 15* (2006), pp. 82–90.
11. Luo, P. A Human-Computer Collaboration Paradigm For Bridging Design Conceptualization and Implementation. In *Proc. of DSV-IS'94* (Carrara, June 8-10, 1994). Focus on Computer Graphics Series. Springer (1995), pp. 129–147.
12. Molina, A.I., Redondo, M.A., and Ortega, M. A methodological approach for user interface development of collaborative applications: A case study. *Science of Computer Programming 74*, 9 (July 2009), pp. 754-776.
13. Sousa, K., Mendonça, H., Vanderdonckt, J.: Towards Method Engineering of Model-Driven User Interface Development. In *Proc. of TAMODIA'2007*. LNCS, Vol. 4849. Springer-Verlag, Berlin (2007), pp. 112-125
14. The Moskitt Environment, ProDevelop, Valencia, 2010. Accessible at http://www. moskitt.org/eng/proyecto-moskitt/
15. Vanderdonckt, J. Model-Driven Engineering of User Interfaces: Promises, Successes, and Failures. In *Proc. of ROCHI'2008 (Iasi, September 18-19, 2008)*, S. Buraga, I. Juvina (eds.). Matrix ROM, Bucarest, 2008, pp. 1–10.
16. Wurdel, M., Sinnig, D., Forbrig, P.: Task-Based Development Methodology for Collaborative Environments. In *Proc. of EIS'2008*. LNCS, Vol. 5247. Springer, Berlin (2008), pp. 118-125.